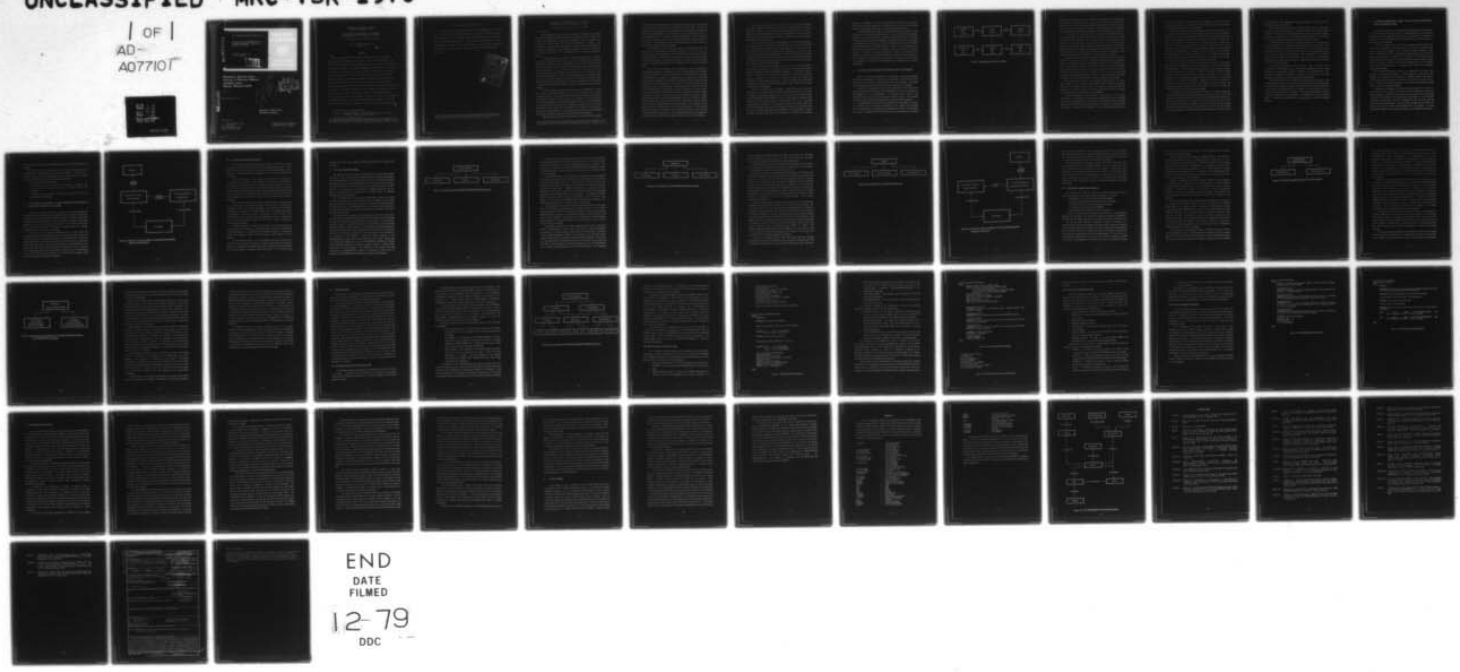
AD-A077 101 WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER F/G 5/2
THE DESIGN AND IMPLEMENTATION OF A DATABASE MANAGEMENT SYSTEM U--ETC(U)
JUN 79 A J BAROODY, D J DEWITT DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-1970 NL

| OF |
AD-
A077101



END
DATE
FILMED
12-79
DDC

MRC TECHNICAL SUMMARY REPORT # 1970

THE DESIGN AND IMPLEMENTATION OF A DATABASE MANAGEMENT SYSTEM USING ABSTRACT DATA TYPES

A. James Baroody, Jr.
David J. DeWitt

LEVEL II

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

June 1979

Received May 29, 1979

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D.C. 20550

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

THE DESIGN AND IMPLEMENTATION OF A DATABASE MANAGEMENT SYSTEM USING ABSTRACT DATA TYPES

A. James Baroody, Jr. and David J. DeWitt

Technical Summary Report #1970
June 1979

-A-

ABSTRACT

The design, implementation, and performance analysis of a database management system implemented using abstract data types are presented. The use of abstract data types as an implementation tool is shown to have several significant advantages over current implementation techniques. First, by using a combination of abstract data types and generic procedures to structure the design of a database system, the resulting software will be more reliable. Also, by employing a programming language which supports specification and verification of abstract data types, we can guarantee data independence. Finally, we demonstrate that the application of abstract data types permits the elimination of run-time interpretation of the schema and subschema such as in IBM's IMS, Univac's DMS110, and INGRES. Instead, the data manipulation routines, which are shown to be examples of generic procedures, are implemented as parameterized calls to the procedures bound to the instances of the three abstract data types used to represent the logical structure of the database.

AMS(MOS) Subject Classification 68A50

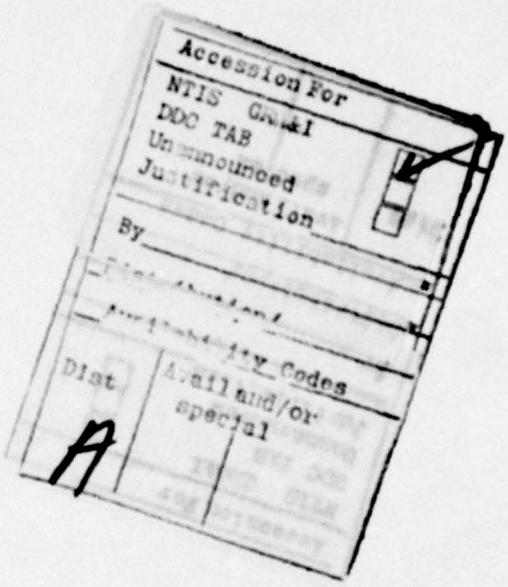
Key Words: Database management systems, abstract data types, generic procedures, CODASYL, query execution

Work Unit No. 8 - Computer Science

This research was partially sponsored by the National Science Foundation under grant MCS78-01721 and the United States Army under contract No. DAAG29-75-C-0024.

Significance and Explanation

A new approach for database management system implementation is presented. This new approach utilizes abstract data types, a feature of modern programming languages such as CLU, ALPHARD, SIMUCA, MODULA, and the new Department of Defense programming language DOD-1. There are several consequences of this implementation technique: more reliable software, guaranteed data independence, and increased run-time efficiency compared to conventional implementation techniques.



The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the authors of this report.

THE DESIGN AND IMPLEMENTATION OF A DATABASE MANAGEMENT SYSTEM USING ABSTRACT DATA TYPES

A. James Baroody, Jr. and David J. DeWitt

1. INTRODUCTION

Data independence requires that user programs be isolated from the details concerning the underlying physical and logical structures used to implement the database. Data independence is achieved by presenting to the user an abstraction of the database in which details of its actual implementation are hidden. The development of tools to support abstraction is an active area in current research on programming methodology and programming languages. This report analyzes the use of abstract data types as an implementation tool for database management systems.

The data model defines a collection of logical structures which are available to the users of a database management system. In order to utilize these structures for applications, the structure of the actual database must be defined in terms of the structures in the data model. The definition of the database is termed the schema, or data model definition. The schema defines the types of entities which may exist within the database and also defines the relationships which may exist between these entities. The schema thus is a definition of the logical structure of a database and is shared by all users of the database. A subschema, or data submodel definition, is the definition of the subset of the database to be used by an application. The subschema is a restriction of the schema and defines an application-dependent window into the database.

The data manipulation routines handle all accesses to the database. A request from a user to access the database is in terms of a procedure call to a data manipulation routine. This procedure uses the schema and subschema to determine the necessary operations to be performed on the database. In the network data model the actual parameters to the data manipulation routines are records and sets. In our approach the record and set are considered to be generic types. The data manipulation routines utilize the information in the schema and subschema to describe the actual parameters in order to determine the function to be performed. Thus the data manipulation routines are examples of generic procedures.

Abstraction arises in the design of data structures from the need to recognize

This research was partially sponsored by the National Science Foundation under grant MCS78-01721 and the United States Army under contract No. DAAG29-75-C-0024.

the similarity between objects and to concentrate on those properties that are shared by many objects while ignoring the differences between them. The use of abstraction in the development of data structures was examined by Hoare [Hoa72]. Hoare applied abstraction to sets of objects to create *discriminated unions.* A discriminated union is a set defined to be the union of two or more previously known sets. Since two sets may have members of different component types, a discriminated union provides a method to distinguish the type of the member by means of a tag associated with each member.

Minsky [Min76] utilizes abstract data types to extend and model the file concept used in traditional data processing applications. A file is viewed as an abstract data type which possesses as one of its attributes the record space of the file in secondary storage. It also contains an attribute, the global space, which contains information describing the characteristics and status of the file as a whole. Bound to these data attributes are procedures which manipulate the record space and global space.

One of the most significant works on abstraction and its relationship to modeling the information in a database is the work of Smith and Smith on relational databases [SS77a] [SS77b]. This work formalizes the use of foreign keys described by Codd [Cod70] as an *aggregation.* An aggregation is an abstraction of a relationship between objects. Aggregation allows details concerning the objects themselves to be ignored when the relationship is being analyzed. *Generalization* abstracts the properties of objects within the database. A generalization is an abstraction in which a set of similar objects is regarded as a collection of instances of a *generic object.* That is, the differences between individual entities are ignored and their common properties are identified and used to classify the individual objects as a single, named, generic object. By explicitly naming generic objects, it is possible to identify generic operators for the generic objects, to specify the attributes of generic objects, and to specify the relationships between generic objects. The properties of each generic type may be formalized by a set of invariant properties. These properties should be satisfied by all relations in the database and should remain invariant following operations on the database. The concepts of generalization and abstraction are also used by Smith and Smith to support different user views of the database.

The objective of this paper is to study the use of programming languages which support abstract data types as a tool for the implementation of database management systems. In our approach abstract data types are used both as an implementation technique similar to the approach of Minsky and also as a database modeling technique incorporating the concepts of aggregation and generalization developed by the Smith and Smith.

In this paper the design and implementation of a network model database management system will be examined. Date [Dat76] and Tsichritzis [Tsi75] [Tsi76] have independently developed programming languages which support the coexistence of the three major data models. These languages are based on a system model very similar to the network data model. Since the network model can be used to implement the relational model and the hierarchical model, solving the problems of implementation for the network data model will result in a solution which can be generalized to include all three data models.

Section 2 describes implementation techniques in current database systems. The generic procedure model is introduced in Section 3 as a technique for describing database management system implementation techniques. This model is used to describe the binding of the actual parameter descriptors to the data manipulation routines. In Section 4 this model is modified to represent the schema and subschema as shared abstract data types. Section 5 analyzes the performance enhancement provided by utilizing compile-time binding rather than run-time binding through interpretation.

There are several major advantages to representing the schema and subschema as shared abstract objects. First, by using abstract data types to structure the design of a database management system, the resulting software should be more reliable [ICRS75] [Kos76] [Lin76] [Wor77]. By using a programming language which supports specification and verification of abstract data types, we can verify data independence at compile-time. As described by Brodie and Tsichritzis [BT77], if specification techniques are employed for defining the schema and subschema, then the user environment presented by the subschema can be guaranteed to remain constant even though the schema and subschema are modified as the database

changes. In addition, it is shown in [Bar78] that abstract data types and programming language support for environment control allow the programming language to support data independence and subschema functions directly.

Another significant advantage of our approach will be an improvement in the performance of a database management system which is implemented using abstract data types. This is possible because application of abstract data types to a database management system permits the elimination of run-time interpretation of the schema and subschema used in systems such as IBM's IMS [McG77], UNIVAC's DMS1100 [SPE75a] [SPE75b] and INGRES [SWKH76]. Instead, data abstractions which represent the logical structure of the database are bound at load-time to the user's program. In addition, the data manipulation routines are implemented by parameterized calls to the procedures bound to the abstract data types which were used to represent the logical structure of the database. In this way we can avoid run-time interpretation of the schema and subschema without suffering any loss of data independence.

2. AN ANALYSIS OF CURRENT IMPLEMENTATION TECHNIQUES

One of the major design decisions in developing a database management system is the determination of when the data manipulation routines bind the data descriptor from the schema and subschema to their actual parameters. In general, the longer binding can be delayed, the easier data independence is to support.

Larson [Lar78] analyzes this binding and considers both *when* and *how* the schema information is bound to the data manipulation routines. The mappings, or transformations of the representations of information, which occur in the development and execution of a user program are shown in Figure 1. The solid lines represent mappings and the dashed lines represent algorithm-data interactions.

The mapping M1 occurs during the development of an algorithm to solve the user's problem. M2 is the process of compiling a source program into executable form. M2 and M4 are very important in the database environment. M2 represents the process of mapping the real-world structures and relationships of the user's

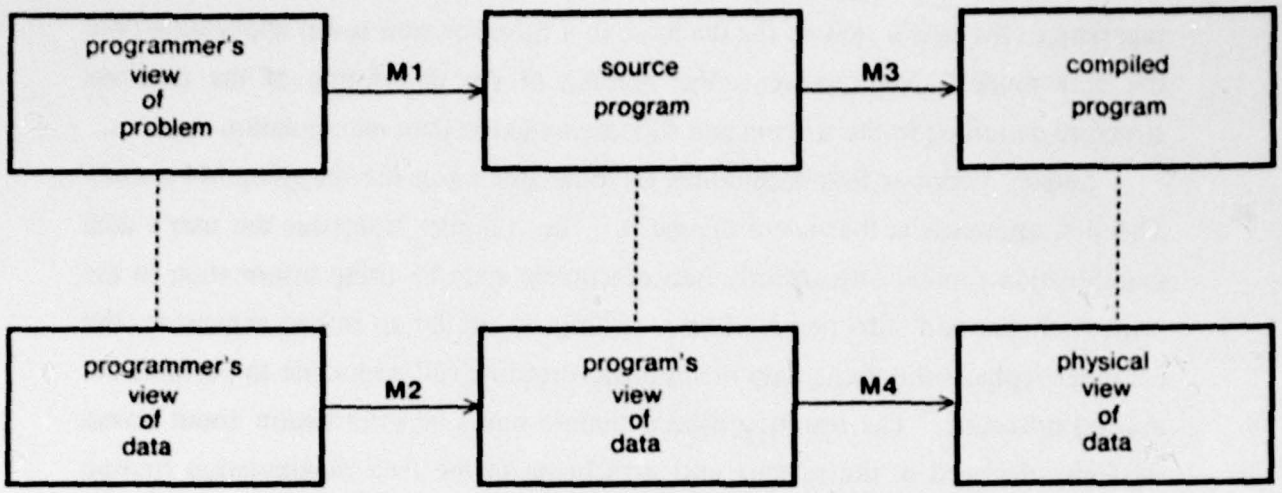


Figure 1 Mappings of the View of Data

application onto the data structures and relationships supported by the data model. This mapping is performed by the schema and subschema which define the mapping of the user's view of the database in a form relevant to his application onto the data model. M4 represents the binding of the description of the database structure described in the schema and subschema to the data manipulation routines.

Larson describes four techniques for how and when the mapping M4 occurs. The first approach is the *macro approach*. The compiler translates the user's data manipulation routine calls directly into executable code by using information in the source schema and subschema. Using techniques similar to macro expansion, the compiler replaces the user's data manipulation routine call with code to perform the desired operation. The resulting object module binds all information about access strategies declared in the schema and subschema to the data manipulation routine calls and the object module thus directly accesses the database at run-time.

This approach has the traditional advantages of compilation, that is, execution efficiency and the facility to utilize multiple programming languages and library routines. An apparent advantage in the database management system application is that no explicit space is required for the object schema and the object subschema. This advantage is perhaps illusory, since in the macro approach the information from the schema is still present but is now distributed in the code, rather than being isolated into an encoded representation. However, the generated code is optimized for space since it only contains procedures explicitly referenced in the schema.

These advantages are offset by several significant disadvantages. The first is that multiple users imply multiple object modules. Each of these object modules contains code to manage concurrent access to the database. This decentralization of concurrency control makes control of concurrent access more difficult than if a single resource manager were controlling access to the database.

Another disadvantage of this approach is its potential for loss of data independence. Any modification of the schema or subschema will require all object modules using the database to be regenerated. More importantly, after combining the schema and subschema with the user program it is difficult to guarantee that the user program is isolated from knowledge of the access methods declared in the

schema and subschema. Coupled with these problems is the fact that few programming languages have the capability to define structures as complex as those that exist within a database management system.

Rather than binding all information from the schema and subschema to the user program at compile-time, data independence suggests that this binding be delayed. In the *library approach* binding of the schema and subschema to the data manipulation routines occurs partially at compile-time and partially at run-time. The user's data manipulation routine calls are translated into calls to a library of procedures. This library is then bound to the user program at load-time. This approach is similar to an I/O library in which language operations such as READ, GET, PUT, and WRITE are translated into calls to routines in an I/O library. Parameters required by the library procedures, such as file name, record length, block length, device on which the file is located, etc., are supplied from the user program, from the job control language, or from file control blocks.

A mechanism which is similar to the library approach is utilized in Burroughs' DMSII [BUR75]. DMSII uses two phases of compilation. During the first phase the data manipulation routines, the schema, and the subschema are compiled to produce an object module which is a library, or collection, of procedures called the access methods. In the second phase, the user program is compiled. The schema and subschema are accessed during compilation to perform type checking for the entities declared in the schema and subschema. Binding of the user environment is completed at load-time by binding the access methods to the user program. By separating the access methods from the user program, it is possible to modify the access methods without affecting the user programm.

The most frequently-used implementation technique is to postpone binding the schema and subschema to the data manipulation routines until run-time. This is the *interpretive approach* and variations of it are used in IMS, DMS1100, INGRES, etc. The schema and subschema are encoded into an internal form, referred to as the object schema and the object subschema. Using the record types and set types referenced as actual parameters in a data manipulaton routine call, the object schema and object subschema are accessed to retrieve the appropriate record type and set

type descriptors. These descriptors are then interpreted to perform the data manipulation language command.

The interpretive approach has some significant disadvantages. Interpretation will require increased processing time to interpret the object schema and the object subschema. If the object schema and subschema are stored on secondary storage to facilitate their being shared by all programs accessing the database, then increased I/O costs will be incurred in addition to the increased processing costs.

In DMS1100, which we believe is a typical example of this approach, three phases of binding are used. The first phase translates the source schema into a form suitable for interpretation at run-time. A secondary output of the first phase is a set of specifications for a user work area which will serve as a communications buffer between the user program and the database. The user program is then compiled using the object schema for type-checking. The user work area specifications generated in the previous phase are used as a template to generate a user work area within the users object module.

The third, and final, phase occurs at run-time. Each user call to a data manipulation routine specifies one or more actual parameters. For each actual parameter the corresponding descriptor is located in the encoded schema. This descriptor is interpreted to determine the function to be performed by the data manipulation routine, e.g. the access methods to be employed, etc.

The fourth approach is the *automatically-generated procedure approach*. The data manipulation routine calls are treated as calls to procedures which are generated at the time that the database is opened. Larson describes this approach as similar to a host computer generating code to be executed by a slave computer, such as a channel when performing an I/O operation. In the general database case, the host and slave computer are the same computer. This approach is in the development stage. An example of a system similar to this approach is described by Stemple [Ste76].

3. A GENERIC PROCEDURE MODEL OF DATABASE MANAGEMENT SYSTEM ARCHITECTURES

A generic procedure is a procedure which performs the same basic operation on actual parameters of more than one type. The implementation of the operation will generally vary depending upon the type of the actual parameter. The plus (+) operation in ALGOL 60 represents such a procedure since it accepts operands of types integer, real, complex,etc.

In order to implement all of the various computations, the procedure must have access to a description of the characteristics of all valid actual parameter types. Thus the types of the operands can be regarded as implicit parameters to the operator. The descriptor of the actual parameter characteristics may be bound to the generic procedure either at compile-time or at run-time. If the binding occurs at compile-time, then the descriptor is the type information from the compiler symbol table. For example, an instance of the procedure implementation may be supplied for each type of actual parameter. The procedure body instances are expanded at compile-time as macros.

If the binding is performed at run-time, two approaches can be used. One approach is to perform interpretation. Another approach is illustrated by EL1 [Weg74]. EL1 represents type information as a MODE (similar to the tag of a variant record) which is associated with each variable and which is testable at run-time. A procedure may perform operations on its arguments depending upon the run-time value of the argument's MODE.

A database can be regarded as a collection of record types with variants, or a discriminated union of records. In this context the data manipulation routines are generic procedures. The tag of each record is used to determine the record's type and hence the computation to be performed by the procedure. The schema and subschema associate with a record the specifications of the data items plus the specification of an access strategy. The access strategy specifies the procedures to be used to store and to retrieve record instances from the database. The data manipulation routines thus utilize these schema and subschema descriptors to

determine what functions are to be performed for a given actual parameter to the procedure.

By representing the data manipulation routines as generic procedures, we can model a database management system as shown in Figure 2. This model of a database management system demonstrates that the implementation techniques examined earlier differ in terms of the following characteristics:

1. Time of binding the data descriptor
2. Mechanisms employed to share the data descriptors among all user processes accessing the database through a particular schema and subschema
3. Controlling access to attributes within the data descriptor to enforce or guarantee data independence.

4. A DATABASE MANAGEMENT SYSTEM ARCHITECTURE BASED ON THE USE OF ABSTRACT DATA TYPES

The generic procedure model also provides a convenient framework for unifying research on the application of programming methodologies and abstraction techniques to the design of database management systems. In Section 3, the generic procedure model was introduced in order to describe the binding of the actual parameter types to the data manipulation routines. In this section, abstract data types are analyzed in terms of the generic procedure model as a database management system implementation technique.

In Section 4.1 two apparently conflicting functions of the schema and subschema are introduced and discussed. Then, in Section 4.2, we discuss the use of abstract data types as a technique for resolving the conflicting functions. Section 4.3 describes the features a programming language must provide in order to support the abstract data type model. In Section 4.4 we describe the implementation of the abstract data types which represent the schema and subschema. Finally, in Section 4.5 we demonstrate that the data manipulation routines can now be implemented solely in terms of the procedure attributes of the abstract data type instances which are used to represent the schema and subschema.

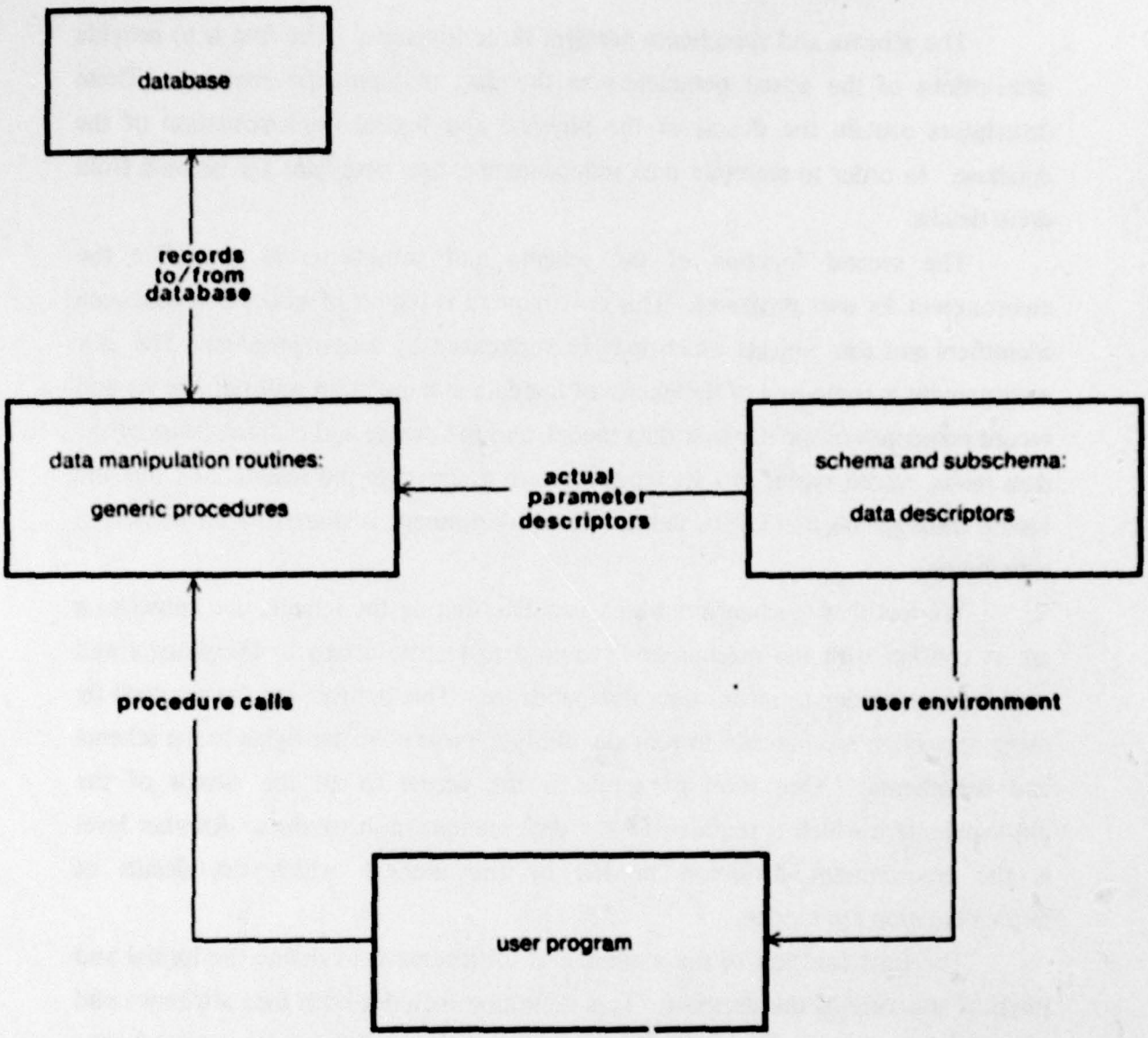


Figure 2 A Generic Procedure Model of Database Management System Architectures

4.1 The Schema and Subschema Functions

The schema and subschema perform three functions. The first is to provide descriptions of the actual parameters to the data manipulation routines. These descriptors contain the details of the physical and logical implementation of the database. In order to maintain data independence, user programs are isolated from these details.

The second function of the schema and subschema is to define the environment for user programs. This environment is the set of associations between identifiers and data objects which may be referenced by a user program. The user environment is composed of the names of the data manipulation routines, the set and record constructs of the network data model, and the names and characteristics of the data items, record types, and set types that are declared in the schema and that are visible through the user's subschema. This environment is shared by all users of a subschema.

We feel that mechanisms which facilitate sharing the schema and subschema are in conflict with the mechanisms required to restrict access to the schema and subschema in order to insure data independence. This conflict can be resolved by using protection mechanisms to provide multiple levels of access rights to the schema and subschema. One level of access is the access to all the details of the implementation which is required by the data manipulation routines. Another level is the environment definition needed by the user in which the details of implementation are hidden.

The final function of the schema and subschema is to define the logical and physical structure of the database. This definition includes both data attributes and procedural attributes. For example, the access strategy defined for a record type specifies the procedure to used to store and retrieve record instances from the database.

In the following section we will demonstrate that the record and set constructs of the network data model can be naturally represented by three abstract data types. The schema and subschema for a given database are thus transformed into a shared collection of instances of these three types. As we will show, this

approach easily resolves the conflicting requirements placed upon the schema and subschema.

4.2 The Three Abstract Data Types

The database is a resource which is shared by all users. Instead of the database being local to a user's address space, the database resides in a separate address space. The user program must execute explicit primitives to move record instances between the database and the user's local address space. A user work area (UWA) serves as a communications buffer for all record instances transferred between the user and the database. As a consequence of this separation of address spaces, there are two different representations, or viewpoints, of the database which the database management system must support.

The first representation is the physical database which is shared by all user programs. Associated with a record instance within the database are the logical pointers which the data manipulation routines use to traverse the database and which are hidden from users. Also associated with each record instance in the database are the encoded forms of the user-visible data items which are defined in the record declaration in the schema.

The database can be regarded as a discriminated union of records. The record type and hence the associated properties of each record instance can be determined by testing at run-time the tag associated with the record instance. By analyzing the common properties of records as they appear in the database we can abstract their properties to define an abstract data type, or generalization, the MassStorageRecord. This abstract data type represents an abstraction of record instances in the physical database. It also provides an abstraction of the participation of a record instance in set occurrences in the database. Using the schema, instances of the MassStorageRecord are generated for each record type. For example, abstract data types MSPresident, MSState, etc. from the PRESIDENTIAL database are defined using the MassStorageRecord as a template as shown in Figure 3 . Details of this abstract data type and examples of the definition of the abstract data types representing the PRESIDENTIAL schema are described in Section 4.4.1 (a description of the PRESIDENTIAL database can be found in Appendix A).

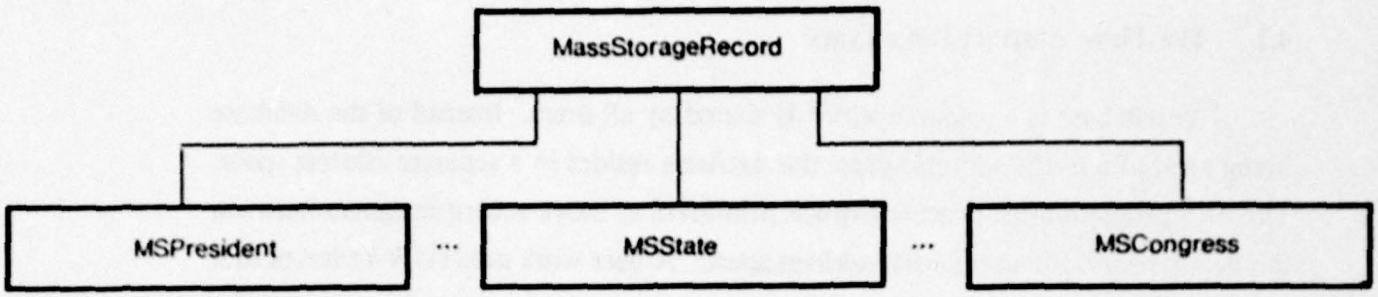


Figure 3 An Example of the MASS STORAGE RECORD Hierarchy

A second view of the database is that view provided to the user through the User Work Area (UWA). The UWA provides the user a view of the record and set types defined in the schema which are visible through the user's subschema. The concept of a record and a set in the network data model can be abstracted to define two abstract data types. Instances of these two types can be used to define the user's view of the database.

The first abstract data type, the UWARecord, is used to generalize the common set of properties possessed by all record types. The UWARecord is an abstraction of the network data model record type. Associated with the UWARecord are data attributes and procedures which represent the procedural information contained in the location mode clause. Additional procedures are associated with the UWARecord which map the user's view of a record type to/from its corresponding representation as a MassStorageRecord in the database. The UWARecord abstract data type is used to define instances corresponding to each record type in the schema. An example from the PRESIDENTIAL schema is shown in Figure 4. The UWARecord generalization is examined in detail in Section 4.4.2.

The second abstract data type represents relationships between record types. This is the UWASet and represents the properties common to all set types. Associated with the UWASet are attributes which describe the record types, that is, MassStorageRecord instances and UWARecord instances, which participate in the set as owner and member. The attributes of the UWASet also include procedures which implement the procedural aspects of the set declaration, such as the set occurrence selection criterion.

The UWASet is related to the concept of aggregation examined by Smith and Smith. (An aggregation is a generic object which represents a relationship between two relations). A UWASet instance actually defines two relationships. The first is between MassStorageRecord instances which participate in set occurrences and is implicitly represented by the procedures associated with the UWASet. The second is the relationship between UWARecord instances which represent the set owner and set member. This relationship is explicitly represented by an owner UWARecord and a member UWARecord pointer in the UWASet instance. The procedures

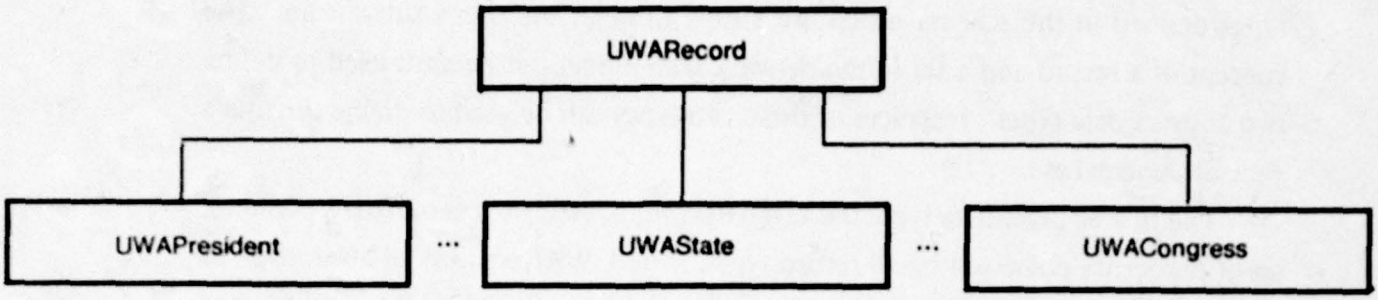


Figure 4 An Example of the UWA RECORD Generalization

associated with the UWASet abstract data type map between the relationship between UWARecord instances, which is visible to users, and the relationship between MassStorageRecord instances, which is not visible to users.

The UWASet abstract data type is used to generate instances corresponding to the set types defined in the schema and subschema. An example is shown in Figure 5. The UWASet is described in more detail in Section 4.4.3.

By defining the schema and subschema as abstract data types, the schema and subschema have been converted from a passive role in the generic procedure model to an active role. We showed earlier that the data manipulation routines are generic procedures. The procedures associated with the UWARecord and the UWASet abstract data types are also generic procedures. Furthermore, as will be shown in Section 4.4, these generic procedures can be regarded as primitives to be used in the implementation of the data manipulation routines. Rather than using a descriptor associated with its actual parameter, the data manipulation routine uses its actual parameter to indirectly call the correct procedure instance. This is shown diagramatically in Figure 6.

Using this approach data independence is achieved in the following manner. Abstract data types provide the capability to restrict access to their attributes. The environment of the user and the environment of the data manipulation routines are two different protection domains. Each domain has a different set of access rights to the schema and subschema. The user is only able to see an environment consisting of the data manipulation routines and the data attributes of the UWARecord and UWASet instances corresponding to the record and set types defined in the schema and subschema. In the environment of the data manipulation routines, however, the operators associated with the UWARecord and UWASet are visible. The MassStorageRecordS corresponding to each UWARecord are also visible. In this way the conflict between data independence and sharing of the schema and subschema which was described earlier can be resolved.

All users of a database logically share the abstract data type instances representing the schema and subschema as well as sharing the database itself. The data attributes of these abstract data type instances represent the status of the user's

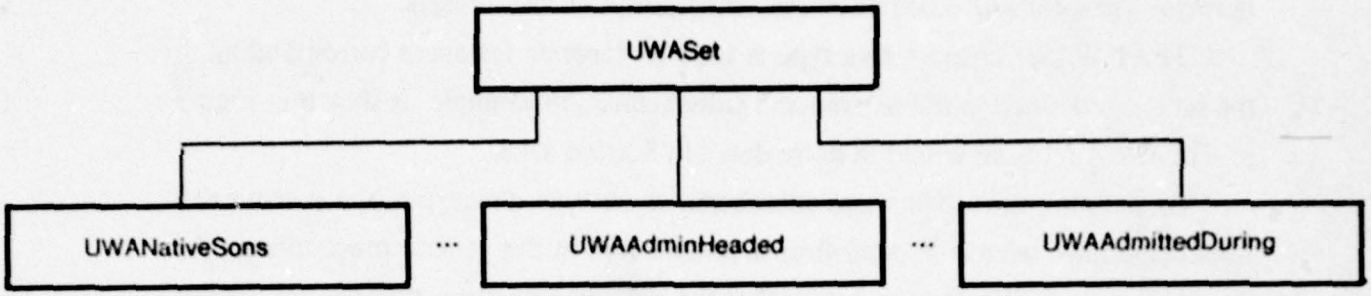


Figure 5 An Example of the UWA SET Hierarchy

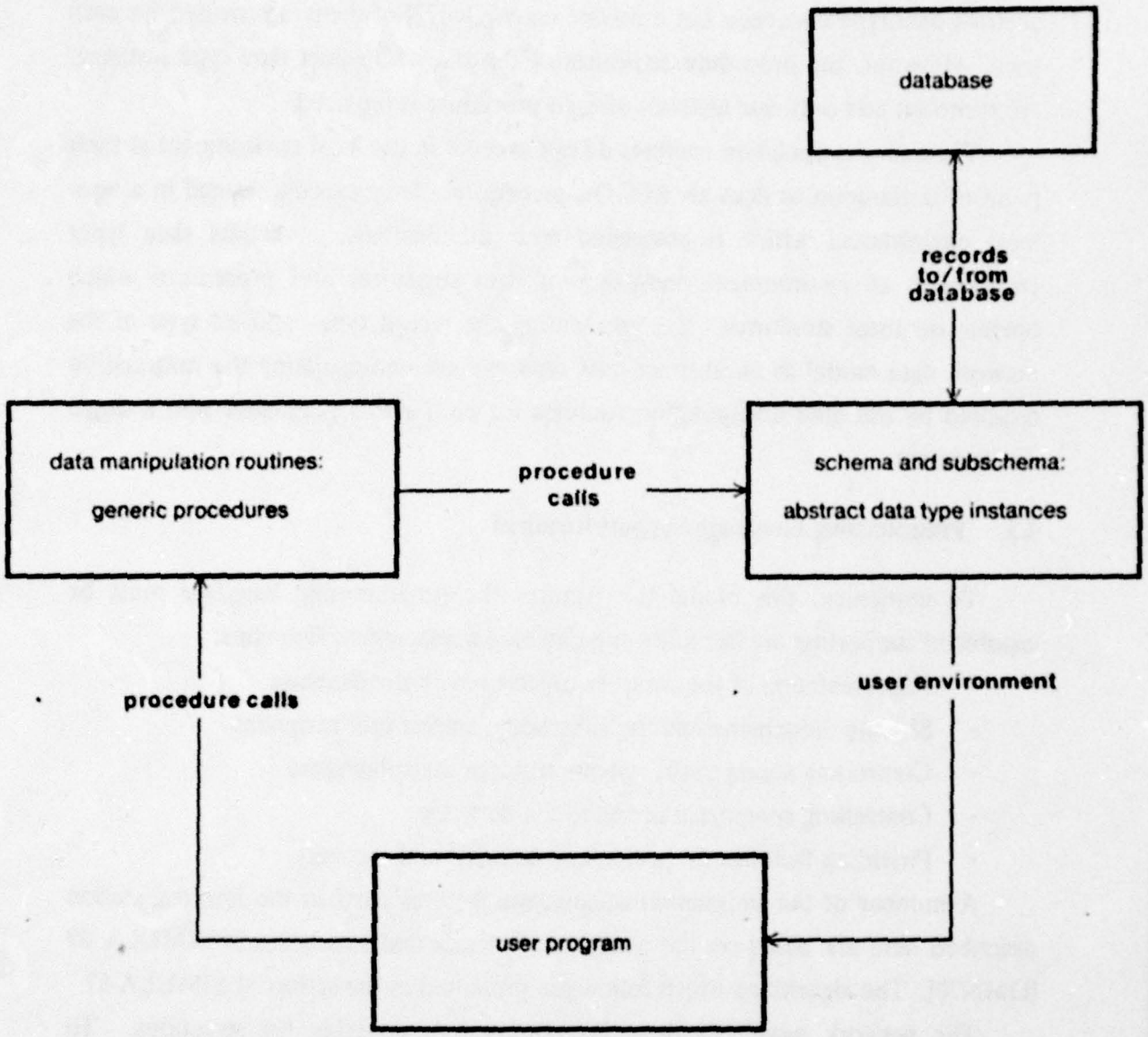


Figure 6 An Abstract Data Type Model of Database Management System Architecture

interaction with the database. They are logically regarded as part of the shared abstract data type instances, but a private copy [Owi77] of them is provided for each user. However, the procedure attributes of the shared abstract data type instances are reentrant and only one instance of each procedure is required.

The data manipulation routines do not execute in the local environment at their point of declaration as does an ALGOL procedure. They execute instead in a non-local environment which is associated with the database. Abstract data types encapsulate an environment consisting of data structures and procedures which operate on these structures. By representing the record type and set type of the network data model as an abstract data type, we are encapsulating the information required by the data manipulation routines for each actual parameter into a single environment.

4.3 Programming Language Support Required

To implement this model the features the programming language must be capable of supporting are the following database management functions:

- Representation of the complex objects within the database
- Sharing the schema and the subschema among user programs
- Controlling access to the schema through the subschema
- Controlling concurrent access to the database
- Providing facilities for protection, security, and recovery.

A number of the programming language features used in the implementation described here are based on the concepts of classes and subclasses in SIMULA 67 [DMN70]. The algorithms which follow are presented in the syntax of SIMULA 67.

The network model database is composed of complex list structures. To represent these structures the language should support pointer variables, user-defined types, and abstract data types. In addition, in order to support the representation of the record and set constructs by abstract data types, it is necessary to separate the declaration of an abstract data type from its implementation. The external view of an abstract data type provides to its user a view of the generic, or abstract, properties of the object. The declaration of an abstract data type is the definition of these

properties. The external view can be used in the development of any modules which access this particular object.

The implementation of these properties is hidden from view. The function of separating the declaration from the implementation is to isolate the use of the abstract data type from its implementation. Frequently this isolation is even stronger than simply hiding the details of implementation. To strengthen this isolation, the declaration of the properties and behavior of the abstract object are created independently from its implementation. For example, CLU completely separates the process of describing the external, abstract properties of an abstract data type from the process of defining their implementation.

VIRTUAL attributes are a mechanism defined in SIMULA 67 for separating the declaration of an abstraction from its implementation. Using VIRTUAL attributes multiple implementations of a common abstraction may coexist, as shown in Figure 7. VIRTUAL procedures are very similar to forward procedures in PASCAL, except that multiple implementations are allowed, one provided by the schema and another implementation possibly provided by each subschema in our approach.

The procedure attributes of the UWARecord and UWASet abstract data types were declared to be VIRTUAL. When an instance of UWARecord (or UWASet) abstract data type is created to form, for example, a UWAPResident instance corresponding to the PRESIDENT record type, a customized procedure for each VIRTUAL procedure will be generated based on the schema definition of that record (or set) type. Furthermore, by using environment concatenation, each VIRTUAL procedure can be redefined based on the subschema through which a user program accesses the database. The schema is defined as a collection of instances of abstract data types in which each instance may have an independent implementation of the VIRTUAL attributes.

As an example, the UWARecord contains a VIRTUAL procedure LOCATE which implements retrieval of MassStorageRecord instances using the strategy specified in the location mode clause of the schema. Using the UWARecord abstract data type as a template, abstract data type instances are generated which correspond

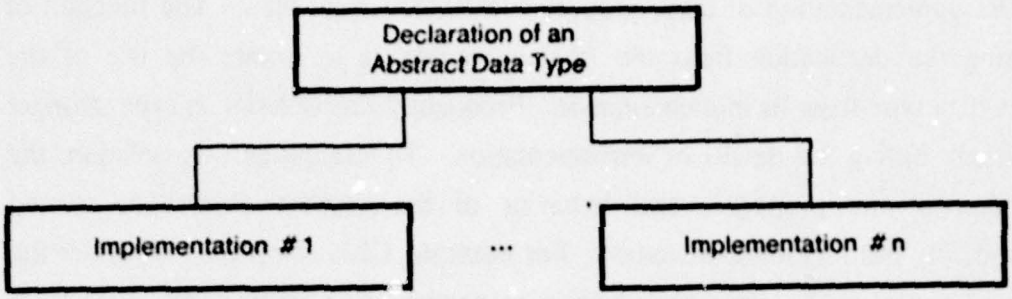


Figure 7 Coexisting Implementations of an Abstraction

to the record types declared in the schema and which implement the particular forms of location mode specified within the record type declarations in the schema. In the PRESIDENTIAL schema, UWAPResident is an implementation of this record abstraction and possesses as an attribute a procedure LOCATE which implements hashing on its LASTNAME attribute. UWAAdmin is also an implementation of this abstraction, but possesses an implementation of LOCATE which performs retrieval via the logical pointers of the AdminHeaded set type. Each instance of a UWARecord in the schema possesses an independent implementation of LOCATE which is based on the location mode clause of the schema record type declarations (see Figure 8).

Support for sharing the schema and subschema and for controlling access to the schema through the subschema appear to be conflicting goals. However, they are both related through the use of environment control facilities. Two very different environment control capabilities are necessary to support the schema and the subschema. The first of these concepts is environment concatenation. To make both the schema and the subschema environments available to the user data manipulation language program, the programming language used to implement the database management system must support the ability to concatenate the environment defined by the schema with the modifications to this environment which are defined by the subschema. This concatenated environment is then available as a "global" environment to the user's data manipulation language program.

The data attributes of the schema represent the global status of the database. In contrast to SIMULA, only one instance of the schema exists and all of its attributes are assumed to be shared. To represent the data which is local to a user, such as the data items associated with the UWARecord instances, private variables are used. A private variable is a variable which is assumed to be part of a shared abstract data type. But a private copy of the variable is generated for each user of the abstract data type.

Two major programming language concepts needed to support the relationship between the schema and subschema are binding and environment, or name scope, control. Binding is the establishment of a value to be associated with an identifier.

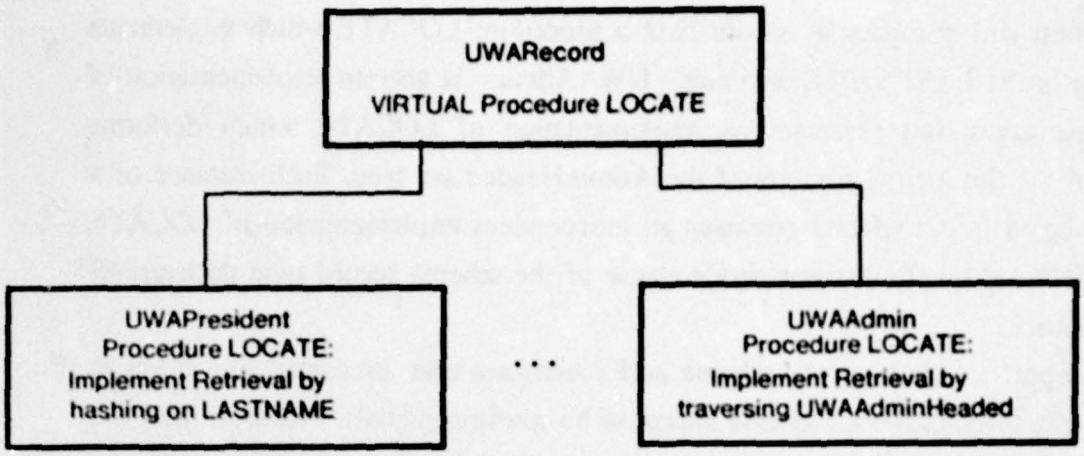


Figure 8 Coexisting Implementations of the UWA RECORD Instances in the PRESIDENTIAL Schema

In the case of the schema or the subschema, binding a record type means associating the entire declaration of a record type with the identifier for the record type supplied in the schema or subschema.

Block structure with the concept of parallel inner blocks is very close to the hierarchical structure of the schema and multiple subschemas. Environment concatenation allows the environment of the outermost block, the schema, to be concatenated with the environment defined by each of the subschemas. This combined environment, with compiler support for separate compilation, can be made available as the enviroment for the data manipulation language program. Thus the data manipulation language programs sharing the subschema represent another level of blocks which are nested within the subschemas so that several user programs can share the same schema and subschema.

What programming language features are required to implement the subschema functions? The ability to define a new identifier to represent an area, set type, or record type requires that the language support the ability to define aliases. Given this capability, the subschema can declare a new identifier of the desired area, record, or set type. The definition of an alias is a command to the compiler to create a second name for an object or data item. All references in the subschema are required to reference the object using the alias. This alias definition may also supply a new data type to be associated with a data item. A reference to the data item thus requires a coercion to be invoked.

The remaining subschema functions are more difficult to support. Eliminating visiblility of identifiers in the subschema can be implemented using several approaches. One approach is a modification of the concept of environment concatenation. As described earlier, environment concatenation follows the ALGOL 60 name scope rules which make all identifiers visible in the outer block visible within an inner block. The environment which a block may pass to its descendants is the environment it inherited from outer blocks plus the locally-defined environment.

This basic function of environment concatenation is in strong conflict with the desire to restrict access to an identifier. The problem in controlling access rights is

to provide a means to restrict the visibility of identifiers which are inherited from a concatenated enivornment or which are locally defined. The HIDDEN specification [Pal76] allows a block to restrict the environment which it passes to inner blocks. A HIDDEN specification may be applied to either an identifier inherited from an outer block or may be applied to a locally-defined identifer. An identifier which is HIDDEN is invisible in inner blocks. We wish to restrict at the subschema level those identifiers which are declared in the schema. To accomplish this we declare at the subschema level that an identifier declared in the schema is HIDDEN and is not accessible to user data manipulation language programs using the subschema. An alternative approach is based on the concept of EXPORTED and IMPORTED declarations which are used in EUCLID [LHLMP77], MESA [GMS77], and TELOS [THLZ77].

The redefinition of the set occurrence selection clause is a more complex problem. It allows the subschema to rebind part of the schema set type declaration. A technique to provide this ability is based on VIRTUAL procedures. Associated with the declaration of the VIRTUAL name is a declaration of the set occurrence selection clause which is associated with the set type declaration in the schema. Subsequently this declaration of the set occurrence selection clause may be replaced by a declaration in the subschema which is bound to the VIRTUAL name. If no redefinition occurs in the subschema, the schema version is used.

4.4. An Implementation

In this section an implementation of the abstract data types which represent the network data model is described. We will describe the implementation of the schema. The subschema is implemented using the same approach.

The schema definition is written in the Data Definition Language (DDL) described in Appendix A. To represent the schema as a collection of abstract data type instances, this DDL schema declaration is translated into the equivalent declaration of a collection of abstract data type instances using the declarations of the abstract data types described above as templates. For each record type declared in the schema, a declaration of a corresponding MassStorageRecord type and of a UWARecord instance is generated. For each set type declared in the schema a corresponding UWASet instance is also generated. These abstract data type and instance declarations are then compiled and are bound to the user program at *load-time* by using environment concatenation. Each of these abstract data types and instances uses environment concatenation to bind the generic properties of the abstract data types to the specific properties of the record type or set type declared in the schema. Each compiled abstract data type declaration also supplies a potentially unique implementation of the VIRTUAL attributes of the generalization on which it is based. By using the access control features of abstract data types an environment for user programs is defined by the schema and subschema abstract data type instances in which the database structure is hidden. This is an implementation of the automatically-generated procedure approach described by Larson. However, rather than generating the procedures for each user when the database is OPENed, our approach generates the schema and subschema once when the database schema and subschema are compiled.

4.4.1 The Mass Storage Record Abstract Data Type

The MassStorageRecord abstract data type represents records as they exist in mass storage. Associated with a record instance in the database are the pointers which the data manipulation routines use to traverse the database and which are hidden from users.

The MassStorageRecord abstract data type defines two subabstractions. The subabstractions, OwnerMSRec and MemberMSRec, represent the participation of a record instance in set occurrences within the database. The OwnerMSRec subabstraction implements the information describing the ownership of set occurrences by a record instance, while the MemberMSRec subabstraction implements the membership of a record instance in one or more set occurrences.

As shown in Figure 9, MSCongress, which corresponds to the CONGRESS record type in the PRESIDENTIAL schema, is declared to be a set owner. Figure 9 also shows that MSCongPresLink, which corresponds to the CongressPresidentLink record in the schema, is defined to be a set member. The OwnerMSRec and MemberMSRec subabstractions are not mutually exclusive, as shown by MSPresident which is both a set owner and a set member.

The MassStorageRecord, which has no procedure attributes, has the following data attributes:

1. Record Type Information - This field is used to tag each record instance in the database with a type descriptor so that run-time checking can be performed.
2. OwnerMSRec - This component of the MassStorageRecord is used to describe the set types which are owned by this record type. The data attributes of the OwnerMSRec are a FIRST pointer field and a LAST pointer field. These data attributes are VIRTUAL.
3. MemberMSRec - This component of the MassStorageRecord abstract data type is used to describe the set types in which the record participates as a member. The data attributes of the Member Subabstraction are PRIOR, NEXT, and OWNER pointer fields These data attributes are VIRTUAL.

This definition of the MassStorageRecord is used to translate each of the record type declarations in the schema into equivalent abstract data type declarations which are used as templates for creating record instances in the database. Using these type declarations as a templates, environment concatenation is used to combine the properties of the record type declared in the schema with those generic properties

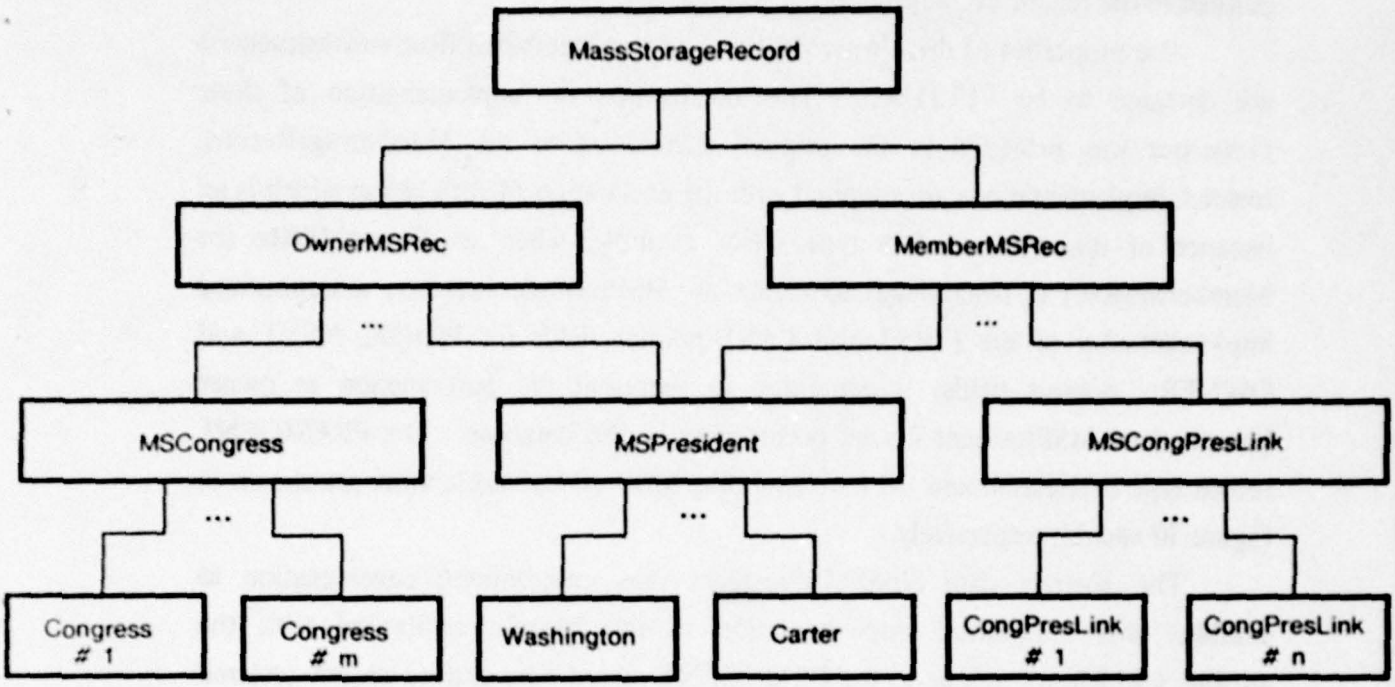


Figure 9 An Example of the MASS STORAGE RECORD Hierarchy

defined in the MassStorageRecord. Concatenated to each MassStorageRecord instance are the possibly encoded forms of the user-visible data items which are defined in the record declaration in the schema.

The properties of the OwnerMSRec and the MemberMSRec subabstractions are declared to be VIRTUAL. This means that no implementation of these properties was provided in the original declaration of the MassStorageRecord. Instead, implementations are supplied with the declaration of each object which is an instance of this abstract data type. For example, when an OwnerMSRec (or MemberMSRec) is instantiated to create an MSPresident instance, a customized implementation of the FIRST and LAST pointer fields (or PRIOR, NEXT and OWNER pointer fields) is generated to represent the participation as owner (member) of MSPresident for set occurrences in the database. The PRESIDENT record type declaration and the corresponding MSPresident declaration are shown in Figure 10 and 11, respectively.

The abstract data type MSPresident uses environment concatenation to combine this customized implementation of the MassStorageRecord with the attributes which are unique to the PRESIDENT record type and which are declared in the schema. Thus a MSPresident abstract data type concatenates to the MassStorageRecord the data items LASTNAME, FIRSTNAME, etc, which are declared in the schema.

4.4.2 The UWA Record Abstract Data Type

The UWARecord abstract data type is an abstraction of the network data model record construct and represents the user's view of records in the database. The UWARecord declaration is a template which is used in creating instances which are part of the user work area. Its data attributes are:

1. Record type information - For run-time checking of record occurrences.
2. CURRENT - Current instance of the MassStorageRecord of this record type.
3. UWASet Instance Pointers - A pointer to all UWASet instantiations to which the record type instantiation is an owner or member. A

```
RECORD NAME IS PRESIDENT
RECORD CODE IS 5
LOCATION MODE IS CALC USING LASTNAME
DUPLICATES ARE ALLOWED
TEXT LASTNAME, FIRSTNAME,INITIAL
TEXT MONTH BORN
INTEGER DAY BORN,YEAR BORN
TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGION
TEXT MONTH DIED
INTEGER DAY DIED, YEAR DIED
TEXT CAUSE OF DEATH, FATHER, MOTHER
```

Figure 10 The President DDL Record Type Declaration

```
MassStorageRecord CLASS MSPresident;
BEGIN
    HIDDEN ALL;

    COMMENT Implement set member pointers ;


    REF(MassStorageRecord) ARRAY NEXT, PRIOR,OWNER (1:2);


    COMMENT Index 1 = member of AllPresidentsSS
            Index 2 = member of NativeSons;

    COMMENT Implement set owner pointers ;


    REF (MassStorageRecord) ARRAY FIRST, LAST (1:3);


    COMMENT Index 1 = Owner of AdminHeaded
            Index 2 = Owner of ElectionsWon
            Index 3 = Owner of CongressServed;

    TEXT LASTNAME, FIRSTNAME,INITIAL;
    TEXT MONTH BORN;
    INTEGER DAY BORN,YEAR BORN;
    TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGION;
    TEXT MONTH DIED;
    INTEGER DAY DIED, YEAR DIED;
    TEXT CAUSE OF DEATH, FATHER, MOTHER;
    COMMENT Initialize Type Information ;
    RCODE := 5; ID := "PRESIDENT";

END;
```

Figure 11 The MSPresident Declaration

UWAPresident Record instantiation of the UWARecord would contain five instances of the UWASet Pointer attribute - one each for the AllPresidentsSS, NativeSons, AdminHeaded,ElectionsWon, and CongressServed set types.

4. NUMSETSMEMBER - The number of sets in which the record type participates as a member.
5. NUMSETSOWNED - The number of sets in which the record type participates as an owner.

The procedure attributes of the UWARecord represent the procedural information contained in the schema. The attributes are:

1. LOCATE Procedure - This procedure is responsible for locating a record instance in mass storage and making it current of record and current of all sets in which it participates. It is a VIRTUAL procedure attribute.
2. ALLOCATE Procedure - This VIRTUAL procedure allocates a new MassStorageRecord occurrence using the MassStorageRecord as a template and the location mode clause from the schema as a storage allocation procedure.
3. LOAD (STORE) Procedure - This VIRTUAL procedure is used to load (store) the items in a UWARecord occurrence from (to) mass storage after the correct MassStorageRecord occurrence is located (allocated).

The definition of the UWARecord is used to translate the record type declarations in the schema into equivalent UWARecord instances. As an example, the UWAPresident instance shown in Figure 13 is generated from the record type declaration shown in Figure 12. Each of the VIRTUAL procedures in the UWARecord declaration is implemented as part of UWAPresident. For example, procedure LOCATE is implemented to perform retrieval of a MSPresident instance by hashing with the LASTNAME attribute as a key.

The local data items of UWAPresident are declared as PRIVATE variables (see Section 4.3). By declaring the local data items of a UWA-RECORD instance as private, they are considered to be part of the shared abstract data type, but an individual copy of these variables is created for each user program using the schema.

```
UWARecord CLASS UWAPresident;
BEGIN
        HIDDEN LOCATE, ALLOCATE, STORE, LOAD;
        PRIVATE LASTNAME,FIRSTNAME,INITIAL,MONTH_BORN,
                DAY_BORN,YEAR_BORN,HEIGHT,
                COLLEGE,ANCESTRY,MONTH_DIED,DAY_DIED,YEAR_DIED,
                CAUSE_OF_DEATH,FATHER,MOTHER,CURRENT;
        TEXT MONTH_BORN,
        INTEGER DAY_BORN, YEAR_BORN;
        TEXT HEIGHT, COLLEGE, ANCESTRY, RELIGIONS;
        TEXT MONTH_DIED;
        INTEGER DAY_DIED, YEAR_DIED;
        TEXT CAUSE_OF_DEATH, FATHER, MOTHER;


        PROCEDURE ALLOCATE;
        COMMENT Allocate a new MSPresident using a hashing algorithm with
        LASTNAME as the key;

        PROCEDURE LOCATE;
        COMMENT Retrieve an MSPresident by hashing with LASTNAME as a key ;

        PROCEDURE STORE;
        COMMENT Perform encoding as required and copy local data items into data
        items of current MSPresident ;

        PROCEDURE LOAD;
        COMMENT Perform decoding as required and copy data items from current
        MSPresident into local data items ;

        COMMENT Initialize type information ;
        RECORDID := "PRESIDENT";
        RECORDCODE := 5;
        NUMSETSOWNED := 3;
        NUMSETSMEMBER := 2;
END;
```

Figure 12 The UWAPresident Declaration

```
SET NAME IS NATIVE_SONS
SET CODE IS 5
OWNER IS STATE
MEMBER IS PRESIDENT
ORDER IS SORTED
ASCENDING KEY IS LASTNAME
DUPLICATES ARE LAST
SET OCCURRENCE SELECTION IS THRU
LOCATION MODE OF OWNER
```

Figure 13 The NativeSons DDL Set Type Declaration

In this way each user program can be guaranteed not to interfere with other user programs.

4.4.3 The UWA Set Abstract Data Type

The second component of the user's view of the database is the set which represents relationships between record types. The UWASet abstract data type represents the properties common to all set types. Associated with the UWASet are attributes which describe the record types, that is MassStorageRecords and UWARecord instances, which participate in the set occurrence as owner and member. The data attributes of the UWASet are:

1. Set Type Information - For run-time checking of set types.
2. CURRENT - Current instance of MassStorageRecord types participating in the set type.
3. OWNER UWARecord Pointer.
4. MEMBER UWARecord Pointer.
5. AUTOMATIC - A Boolean which indicates either manual or automatic set membership.
6. OWNER_OFFSET - The index of the correct FIRST and LAST pointers in the owner MassStorageRecord type.
7. MEMBER_OFFSET - The index of the correct NEXT, PRIOR, and OWNER pointers in the member MassStorageRecord type.

The attributes of the UWASet also include procedures which implement the procedural aspects of the set declaration as follows:

1. INSERT - A VIRTUAL procedure to insert a new member record occurrence into the set using the SET ORDER clause from the schema.
2. LOCATE - A VIRTUAL procedure to locate an occurrence of a set based on the SET OCCURRENCE SELECTION clause of the schema.
3. REMOVE - A VIRTUAL procedure to remove a specific member record occurrence from the set occurrence.
4. SCAN - A VIRTUAL procedure to traverse a set occurrence by following the pointers which link the owner record occurrence and the

member record occurrences and the pointers which link the member record occurrences.

The definition of the UWASet is used to translate the set type declarations in the schema into equivalent UWASet instance declarations. As an example, the NativeSons set type declaration is shown in Figure 13 and the corresponding UWANativeSons is shown in Figure 14 Each of the VIRTUAL procedures of the UWASet declaration is implemented as part of the UWANativeSons declaration.

4.4.4 The Data Manipulation Routines

The data manipulation routines are generic procedures. For example, the FETCH data manipulation routine is a procedure with only one actual parameter, a pointer to a UWARecord instance. Each record (set) declared in the schema provides a detailed description of a record's (set's) characteristics. All of the characteristics of the actual parameter must be known to execute the data manipulation routine.

In our approach, this information is bound to the instances of the UWARecord and the UWASet abstract data types. Rather than perform interpretation, the data manipulation routine uses each actual parameter to indirectly reference the correct data or procedure attribute associated with an abstract data type instance in the schema. Thus the data manipulation routines are completely implemented in terms of the data attributes and procedure attributes of the abstract data types. This use of abstract data types has allowed us to separate the definition of the properties of all databases from those of a specific database. Since the data manipulation routines are isolated from all implementations, they are database independent and data independence is guaranteed.

Figure 15 contains an implementation of the data manipulation routines FETCH. RECPTR references an instance of the UWARecord abstract data type (i.e., a record type).

```
UWASet CLASS UWANativeSons;
BEGIN
        HIDDEN UWA_OWNER, UWA_MEMBER, INSERT, LOCATE, REMOVE, REORDER,
        MEMBER_OFFSET, OWNER_OFFSET;

        PROCEDURE LOCATE;
        COMMENT Implement Set Occurrence Selection through the location Mode of the
        Owner, e.g., call UWAState.LOCATE;

        PROCEDURE REMOVE;
        COMMENT Remove the indicated MSPresident from a set occurrence. Use OWNER
        OFFSET and MEMBER_OFFSET to access the correct pointer locations;

        PROCEDURE SCAN;
        COMMENT Based on the actual parameter switch value move from the current
        record along the correct pointer. Use OWNER_OFFSET and MEMBER_OFFSET to
        access the pointer locations.

        PROCEDURE INSERT;
        COMMENT Insert a new MSPresident instance into a set occurrence. Maintain the
        set order as sorted in ascending order using LASTNAME.;

        COMMENT Initialize type information ;
        SETCODE := 5;
        MEMBER_OFFSET := 2;
        OWNER_OFFSET := 1;
        ID := "NATIVE_SONS";

END;
```

Figure 14 The UWANativeSons Declaration

```
PROCEDURE FETCH(RECPTR);
REF (UWARecord) RECPTR;
BEGIN
        INTEGER I;

        COMMENT use the schema location mode to find the MassStorageRecord and then
        copy  MassStorageRecord fields into UWARecord  fields;

        COMMENT Make the MassStorageRecord the current of record type;

        CURRENTOFRUN :- RECPTR.CURRENT :- RECPTR.LOCATE;

        COMMENT Load all data items into the UWA ;

        RECPTR.LOAD (RECPTR.CURRENT);

        COMMENT Make the record occurrence the current of set in all sets in which it
        participates ;

        FOR     I:=1    STEP    1    UNTIL    RECPTR.NUMSETSOWNED    DO
                RECPTR.OWNER(I).CURRENT :- RECPTR.CURRENT;

        FOR     I:=1    STEP    1    UNTIL    RECPTR.NUMSETSMEMBER   DO
                RECPTR.MEMBER(I).CURRENT :- RECPTR.CURRENT;

END;
```

Figure 15 Data Manipulation Routine FETCH

5. A PERFORMANCE ANALYSIS

As described above the advantages of representing the schema and subschema as abstract data types include increased reliability, improved support for data independence and for sharing the schema and the subschema, and increased run-time efficiency in contrast to the interpretive approach. The increased run-time efficiency is achieved by implementing the data manipulation routines as calls to the procedures associated with the abstract data types representing the schema and subschema. This eliminates the need for run-time interpretation and its associated overhead. In this section we will present a summary of the performance improvement that the abstract data type approach provides in contrast to the interpretive approach. A more detailed discussion of this performance analysis is provided elsewhere [Bar78] [Bar79].

Performance analysis of database management systems has traditionally emphasized the analysis of input/output activity. The size of databases and the need for flexible sharing described earlier have required that the database be stored in secondary memory. The database is in a separate address space from the local address space of the user program. Users must execute explicit commands to transfer record instances between their local address space and the database. Since the database is stored in secondary memory, these commands are actually input/output instructions.

A commonly used implementation technique in current systems is interpretation of the actual parameter descriptors at run-time by the data manipulation routines. This interpretation is performed by the central processor. Since there is a large performance disparity between the access time of conventional storage devices and the cycle time of the central processor, interpretation may not be a significant overhead in the database environment. The objective of this analysis is to characterize the run-time overhead required for interpretation and to relate the elimination of this overhead in the abstract data type approach to system performance.

In a system in which schema interpretation is employed, there are actually

two address spaces, or virtual memories, and their associated buffer pools which are managed by the database management system. The first is the address space of the database in mass storage. The second is the address space in which the encoded schema resides. By generating reference strings to these address spaces and analyzing the page faults which these reference strings produce, the overhead of input/output activity can be derived. These reference strings can also be used to determine the amount of interpretation performed by the data manipulation routines.

Rather than determine these reference strings empirically a combination of simulation and modeling was used. The basic approach was to replace the implementations of the generic procedures associated with the abstract data type instances representing the schema with implementations which simulated I/O to the database and which monitored the points at which interpretation would have been required but was avoided.For example, the procedure LOCATE associated with each UWARecord is replaced with a new implementation which simulates retrieval of a record instance. To study the monoprogramming case, the new implementations were then used to execute synthetic programs individually. To study the multiprogramming case they were also to simulate a mix of synthetic programs in concurrent execution, with each program generating data manipulation routine calls with Poisson-distributed interarrival times. The results of the simulations were a measurement of the number of record requests from the database for each record type and a count of the interpretation of record type and set type descriptors which would have been required.

In order to characterize performance the results of the simulations were used to compute the relative utilization factor, P. P is the ratio of utilization of system resources required to perform interpretation relative to the resources used to perform I/O to the database. Since the user response time is directly related to the these I/O and interpretation costs, measurement of P allows some qualitative predictions about user response time. For example, if P equals 1 and all other costs are assumed negligible, then one-half of the cost of executing a user program is the cost of performing I/O to the database and the remaining cost is interpretation. In this case elimination of the interpretation overhead will reduce user response time by 50% and

double system throughput, unless there are bottlenecks which are hidden by the interpretation overhead.

The simulations measured the frequency of interpretation and the frequency of record requests from the database. Averaged over all of the simulations, interpretation occurred approximately 3.5 times per record request from the database. The simulation model introduced significant simplifications and additional requirements for interpretation of the schema were ignored, such as interpretation of authorization and privacy information. Interpretation of the subschema was not considered. We believe that it is reasonable to expect the ratio of interpretation to record requests is even higher in actual systems than was shown by these simulations.

The simulation results were used to compute the relative utilization. The cost of interpreting a schema descriptor was assumed to consist of two components: The first is the cost to locate the correct descriptor; the second is the cost to evaluate the descriptor. If a descriptor is resident then the cost to locate it is negligible; otherwise an I/O operation is required to retrieve it. Thus P is a function of the CPU cycle time, the mass storage access and transfer times, the size of the schema descriptor and the database buffer pool, the policies used to manage the descriptor and database address spaces, and the multiprogramming level.

We initially consider the monoprogramming case; the multiprogramming case is examined later. The CPU cycle time determines the average time in seconds that will be required to evaluate an schema descriptor. The mass storage access and transfer times determine the average time in seconds which will be required to retrieve or store records in the database or to retrieve a schema descriptor. The absolute values of the CPU cycle time and of the mass storage access and transfer times are not as critical as their ratio, which can be viewed as a measure of the performance disparity between processor speed and mass storage speed. Two configurations were considered. In the first configuration which assumed the presence of conventional mass storage hardware the ratio of average CPU cycle time to the average access and transfer time of a 4K byte page was chosen to be 1:40000. The second configuration assumed the presence of memory hierarchy incorporating CCD devices and thus reduced this ratio to 1:500.

The simulations incorporated management of the descriptor address space by implementing a simulation of a demand fetch policy and a LRU replacement policy. The size of the schema descriptor buffer was chosen large enough to accomodate a working set of schema descriptors. Thus the I/O required to retrieve schema descriptors during execution of the workload was measured directly.

However, I/O to the database could not be directly measured by the simulation. Instead the database I/O was derived from the simulation results by modeling. A database fault occurs only when a requested record is not resident and the requested record (that is, the page which contains it) must be transferred from secondary storage. Empirical data on the locality of database accesses is available from several sources. Schkolnick [Sch77] measured the effect of clustering records within hierarchical systems. Using measurements from IMS, Schkolnick measured page fault probabilities in the range of 0.10-0.20. Similar values were measured by Rodriguez-Rosell [Rod76]. In measurements performed on IMS the page fault probability in the monoprogrammed case was in the range 0.80 < page fault probability < 0.95 and in the concurrent case the page fault probability was less than 0.20.

This empirical data demonstrates dramatic reductions in page fault probabilities due to clustering of related records during traversal of chains within the database. We assume that clustering is used in set implementations but that the randomizing effect of CALCed access using a key value is in contradiction to the aims of clustering. Thus CALCed access should produce a higher number of page faults than traversing a chain within the database which exploits the effects of clustering. To incorporate this effect, we assumed that an I/O operation to the database was required for every third record request.

The values of the relative utilization which were computed were highly dependent upon the characteristics of the actual workloads executed. For the conventional mass storage system the measured relative utilization was in the approximate range of 0.25 to 7.6; for the system which assumed the presence of a memory hierarchy the measured relative utilization was in the approximate range 0.90 to 8.4. These results demonstrate that for the conventional mass storage system

that interpretation accounted for 20%-88% of the system overhead depending upon the user workload. For the higher performance memory system interpretation was 45%-89% of the overhead. These results imply that replacement of slower mass storage devices by higher performance systems may not achieve the anticipated performance gains, but may instead be limited by the interpretation overhead.

The measured relative utilizations were very dependent upon the actual workload. The wide range in the measure value of P is due in large part to the ability (or inability) of a system to maintain a working set of descriptors in the system buffers has a large impact upon performance. One factor relating to this ability to maintain a resident working set is the interval between transitions in the set of descriptors comprising the current working set. This interval is dependent upon the average size of a set occurence, that is the average number of member record instances associated with an instance of an owner record. For traversal of a set occurrence, the larger the set size, the longer the interval between transitions in the working set of descriptors.

A major factor effecting the working set size is the multiprogramming level. To study this effect a mix of the synthetic programs was simulated in concurrent execution, with each synthetic program generating data manipulation routine calls with a Poisson interarrival time. As the level of multiprogramming increased, the availability of schema descriptor buffers became a limiting resource. For an increase in the multiprogramming level one to ten the size of the working set approximately doubled. This results in dramatic increases increases in the relative utilization as the level of multiprogramming increases.

For a small schema, that is, a schema with a small number of record types and set types, it may be reasonable to load the entire schema into memory when the database is opened. But for a large schema, that is, a schema with many record types and set types, this may not be possible. If concurrent users are accessing disjoint subsets of the database, then contention for schema buffers and the resulting increased rate of schema descriptor faults will significantly impact the system performance.

Based on these simulation results for the multiprogramming case we can

make several suggestions for current implementations using interpretation. Due to the dependence of the relative utilization upon multiprogramming local page replacement algorithms should be used to manage the schema descriptor buffers rather than global algorithms. This suggests that subschemas should be designed to reflect the working set size.

The interpretation overhead discussed above is completely avoided when the schema is represented by a collection of abstract data types. In contrast, two different sources of overhead are present. The first is the use of indirect addressing to reference the correct data attribute or procedure attribute within the schema. This overhead is insignificant compared to the overhead of interpretation.

The second overhead which is present at run-time is much more significant. By implementing the data manipulation routines in terms of the procedure attributes of the abstract data types representing the schema, each reference to the schema involves a context switch. Given the overhead of procedure entry and exit, the number of procedure calls which are executed by each data manipulation routine could involve an overhead which is comparable to the CPU time required to perform interpretation. This can be avoided by regarding the data manipulation routines as open subroutines and expanding their defintions inline at the point of call rather than compiling them as closed subroutines.

6. CONCLUSIONS

In this paper we have examined the use of abstract data types as an implementation tool for database management systems. This approach has several significant advantages over current implementation techniques. First, by using a combination of abstract data types and generic procedures to structure the design of a database system, the resulting software should be more reliable. Also, by employing a programming language which supports specification and verification of abstract data types, we can guarantee data independence. Finally, the application of abstract data types permits the elimination of run-time interpretation of the schema and subschema such as in IBM's IMS and Univac's DMS1100. Instead, the data

manipulation routines, which we demonstrated are examples of generic procedures, are implemented by parametized calls to the procedures bound to the instances of the three abstract data types used to represent the logical structure of the database.

Another use of abstract data types which has been proposed is as a very-high-level data model ([HM76], [Ham78], [Min76]). In this approach each entity set is represented by an abstract data type and instances of the entity set correspond to instantiations of the abstract data type. Furthermore, the only operators which can be applied to the database are those bound to the abstract data types which are used to represent the logical structure of the database. In addition to providing a very natural and data independent interface for the user, this approach simplifies the task of enforcing database integrity.

The best way to contrast our use of abstract data types as an implementation tool with their use as a very-high-level data model is to consider each approach in terms of the proposed ANSI/SPARC database framework. The ANSI/SPARC proposal is intended to support views (external models) of the database in terms of all theree currently-used data models: the hierarchical model, the network model, and the relational model. The use of abstract data types as a very-high-level data model can simply be viewed as another external model.

We believe that the abstract data type model which we have presented in this paper is a feasible implementation technique for the ANSI/SPARC architecture [ANS75]. Major questions exist concerning the efficiency required in such a system [BS78]. A number of these questions are resolved by our approach. Using techniques similar to those described in this paper, each external view of the database is represented by a collection of abstract data types. The abstract data types implement a view of the database in terms of their data attributes and procedures. The procedures associated with the abstract data types in a given level of the system are implemented in terms of the data attributes and procedures of levels which are closer to the actual details of implementation. For example, the abstract data types representing the relational view of the database may be implemented in terms of the data attributes and procedures which represent the conceptual level. The abstract data types at the conceptual level are themselves implemented in terms of the data

attributes and procedures of the internal level. We are currently investigating this approach for implementing the ANSI/SPARC architecture.

The abstract data type model which we have described also provides an efficient implementation technique for the revised Data Definition Language (DDL78 [COD78]). In DDL78 the specification of the schema to storage mapping which was formerly part of the schema has been removed and is now defined in the Data Storage Description Language (DSDL). The environment concatenation function of the SIMULA subclass allows us to define multi-level abstract data types. The generic procedures and data attributes of each level correspond to the different schemas. This concatenated object represents the user's view of the record construct. The independence of the concatenated objects is guaranteed by the environment control mechanisms provided by abstract data types. As an example the abstract data type UWARecord defined earlier can be replaced by the DdlUWARecord which implements the attributes defined in the DDL78 schema. The DsdlUWARecord is then defined as a subclass of the DdlUWARecord and implements the attributes defined in the DSDL.

Appendix A

The 1971 CODASYL DBTG Report [COD71] and the CODASYL Data Definition Language Journal of Development [COD73] describe in detail a language, the Data Definition Language (DDL), to be used to describe the logical structures and storage structure of the database. The DDL provides the capability to describe the database in terms of logical subdivisions, or areas, records, and sets. A grammar (BNF) for the DDL which was used in this report and which is based on the CODASYL report is shown below.

```
<SCHEMA>              ::=  <SCHEMA SECTION>
                           <AREA SECTION>
                           <RECORD SECTION>
                           <SET SECTION>
<SCHEMA SECTION>      ::=  SCHEMA NAME IS ID
<AREA SECTION>        ::=  "Not Implemented"
<RECORD SECTION>      ::=  RECORD SECTION
                           <RECORD DECL LIST>
<SET SECTION>         ::=  SET SECTION <SET DECL LIST>
<RECORD DECL LIST>    ::=  <RECORD DECL>
<RECORD DECL LIST>    ::=  <RECORD DECL LIST>
                           <RECORD DECL>
<RECORD DECL>         ::=  <RECORD NAME>
                           <RECORD CODE>
                           <LOCATION MODE>
                           <DUP> <WITHIN> <ITEM LIST>
<RECORD NAME>         ::=  RECORD NAME IS ID
<SET DECL LIST>       ::=  <SET DECL>
<SET DECL LIST>       ::=  <SET DECL LIST><SET DECL>
<SET DECL>            ::=  <SET NAME> <SET CODE> <MODE>
                           <ORDER> <OWNER> <MEMBER>
<SET NAME>            ::=  SET NAME IS ID
<SET CODE>            ::=  SET CODE IS INTNUM
<MODE>                ::=  MODE IS CHAIN
<MODE>                ::=  MODE IS POINTER ARRAY
<ORDER>               ::=  ORDER IS <OC> <SWITHIN>
<OC>                  ::=  FIRST
<OC>                  ::=  LAST
<OC>                  ::=  NEXT
<OC>                  ::=  PRIOR
<OC>                  ::=  SORTED
<SWITHIN>             ::=  WITHIN ID BY ID
<OWNER>               ::=  OWNER IS ID
<MEMBER>              ::=  MEMBER IS <CLASS CODE>
                           <LINKED><MOD><SOS>
<MOD>                 ::=  EPSILON
<MOD>                 ::=  <MORDER> <MDUP>
<MORDER>              ::=  <UPDOWN> KEY IS ID
<MDUP>                ::=  DUPLICATES ARE FIRST
```

<MDUP>	::=	DUPLICATES ARE LAST
<MDUP>	::=	DUPLICATES ARE NOT ALLOWED
<MDUP>	::=	DUPLICATES ARE SYSTEM ORDERED
<SOS>	::=	SET OCCURRENCE SELECTION IS THRU <SOSMODE>
<SOSMODE>	::=	LOCATION MODE OF OWNER
<SOSMODE>	::=	ID USING CURRENT OF SET
<SOSMODE>	::=	ID USING LOCATION MODE OF OWNER
<UPDOWN>	::=	ASCENDING
<UPDOWN>	::=	DESCENDING

Two aspects of the Journal of Development DDL are not considered in this report. The first is the DDL descriptions of the privacy control and integrity constraints which the data manipulation routines are to enforce on the database. The second aspect is the Device Media Control Language (DMCL) which is described in the report as a tool to specify the mapping of the logical database structure onto the physical storage media. While these features of the report are not considered here, we believe that our approach could incorporate them without difficulty.

The examples used in this report are based on the PRESIDENTIAL database defined by Fry and Sibley [FS76]. The schema for the PRESIDENTIAL database is shown in Figure 16.

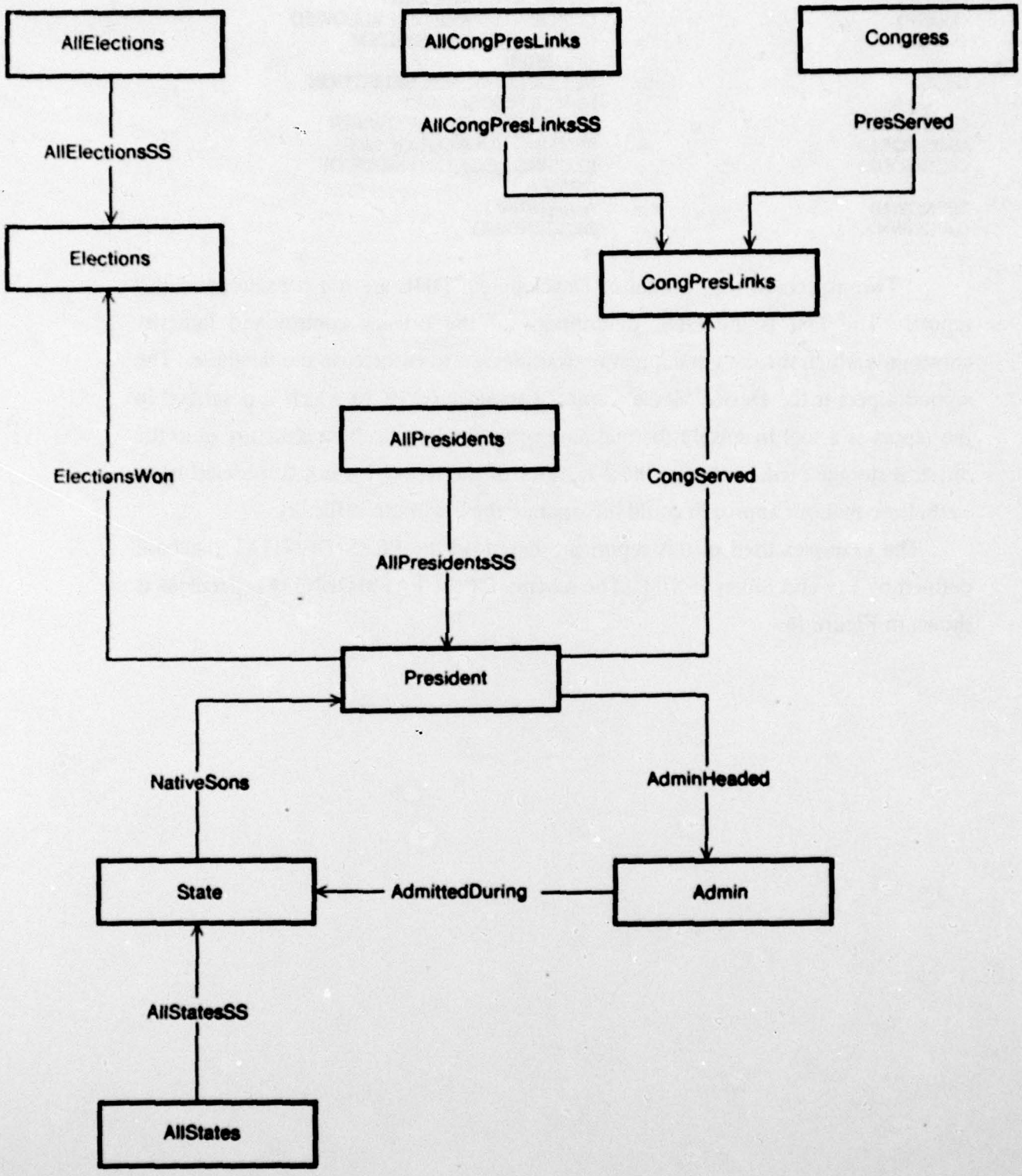


Figure 16 The PRESIDENTIAL Database Schema

REFERENCES

[ANS75] ANSI/X3/SPARC Interim report 75-02-08, FDT BULLETIN OF ACM SIGMOD, Vol. 7, No. 2, February 1975.

[Bar78] Baroody, A. J. Ph.D. Dissertation, University of Wisconsin-Madison, 1978.

[Bar79] Baroody, A. J. In preparation.

[BS78] Brodie, M. L. and Schmidt, J. What is the use of abstract data types in data bases? PROC. INTERNAT. CONF. ON VERY LARGE DATA BASES, Berlin, Germany, pp. 140-141.

[BT77] Brodie, M. L. and Tsichritzis, D. Data base constraints. In A PANACHE OF DBMS IDEAS, Tsichritzis, D. (Ed.), University of Toronto Technical Report CSRG-78, University of Toronto, February 1977, pp. 19-36.

[BUR75] Burroughs Corporation. BURROUGHS B6700/B7700 DMSII DATA AND STRUCTURE DEFINITION LANGUAGE (DASDL) REFERENCE MANUAL. Form No. 5001084, Burroughs Corporation, October 1975.

[COD71] CODASYL DATA BASE TASK GROUP REPORT. ACM, New York. 1971.

[COD73] DATA DESCRIPTION LANGUAGE JOURNAL OF DEVELOPMENT, Document C13.6/2:13, U.S. Government Printing Office, Washington, D.C., 1973.

[COD78] DATA DESCRIPTION LANGUAGE COMMITTEE JOURNAL OF DEVELOPMENT, January 1978.

[Cod70] Codd, E. F. A relational model of data for large shared data banks. COMM. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[DMN70] Dahl, O. J. , Myhrharg, B. , and Nygaard, K. THE SIMULA 67 COMMON BASE LANGUAGE. Pub. S-22, Norwegian Computing Center, Oslo, 1970.

[Dat76] Date, C. J. An architecture for high-level language extensions. ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA. June 1976, Washington, D. C., pp. 101-122.

[FS76] Fry, J. P. and Sibley, E. H. Evolution of data-base management systems. ACM COMPUTING SURVEYS, Vol. 8, No. 1, March 1976, pp. 7-42.

[GMS77] Geschke, C.M., Morris, J. H. , and Satterthwaite, E. H. Early experience with MESA. COMM. ACM., Vol. 20, No. 8, August 1977, pp. 540-553.

[GG77] Gries, D. , and Gehani, N. Some ideas on data types in high-level languages. COMM. ACM, Vol. 20, No. 6, June 1977, pp. 414-420.

[Ham76] Hammer, M. M. Data abstractions for data bases. PROC. OF CONF. ON DATA: ABSTRACTION, DEFINITION AND STRUCTURE. March 1976, pp. 58-59.

[HM78] Hammer, M. M. and McLeod, D. J. The semantic data model: a modelling mechanism for data base applications. PROC. OF SIGMOD INTERNAT. CONF. OF MANAGEMENT OF DATA, Austin, Texas, May 31-June 2, 1978, pp. 26-36.

[Hoa72] Hoare, C. A. R. Notes on data structuring. In STRUCTURE PROGRAMMING by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press, New York, 1972.

[ICRS75] INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE. SIGPLAN NOTICES, Vol. 10, No. 6, June 1975.

[Kos76] Koster, C. H. A. Visibility and types. PROC. OF THE SIGPLAN/SIGMOD CONF. ON DATA: ABSTRACTION, DEFINITION, AND STRUCTURE, March 22-24, 1976, pp. 179-190.

[LHLMP77] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L. Report on the programming language EUCLID. SIGPLAN NOTICES, Vol. 12, No. 2, February 1977.

[Lar78] Larsen, J. A. Personal communication, May 1978.

[Lin76] Linden, T. A. The use of abstract data types to simplify program modifications. PROC. OF SIGPLAN/SIGMOD CONF. ON DATA: ABSTRACTION, DEFINITION, AND STRUCTURE, March 22-24, 1976, pp. 12-23.

[McG77] McGee, W. C. The information management system IMS/VS. IBM SYSTEMS JOURNAL, Vol. 16, No. 2. 1977, pp. 84-168.

[Min76] Minsky, N. Files with semantics. PROC. OF THE ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA, June 2-4, 1976, pp. 65-74.

[Pal76] Palme, J. New feature for module protection in SIMULA. SIGPLAN NOTICES, Vol. 11, No. 5, May 1976, pp. 59-62.

[Rod78] Rodriguez-Rosell, J. Empirical data reference behavior in data base systems. COMPUTER Vol. 9, No. 11, November 1976, pp. 9-13.

[Sch77] Schkolnick, M. A clustering algorithm for hierarchical structures. ACM TRANS. ON DATABASE SYSTEMS, Vol. 2., No. 7, May 1977, pp. 27-44.

[SWL77] Shaw, M., Wulf, W. A., and London, R. L. Abstraction and verification in ALPHARD: defining and specifying iterators and generators. COMM. ACM., Vol. 20, No. 8, August 1977, pp 553-564.

[SS77a] Smith, J. M., and Smith, D. C. P. Database abstractions: aggregation and generalization. ACM TRANS. ON DATABASE SYSTEMS, Vol. 2, No. 2, June 1977, pp 105-133.

[SS77b] Smith, J. M. and Smith, D. C. P. Database abstractions: aggregation. COMM. ACM, Vol. 20, No. 6, June 1977, pp. 405-413.

[SPE75a] Sperry Univac. 1100 SERIES DATA MANAGEMENT SYSTEM (DMS 1100) SCHEMA DEFINITION DATA ADMINISTRATOR REFERENCE MANAUAL, UP-7907, Rev. 2, Sperry Univac, 1975.

[SPE75b] Sperry Univac. 1100 SERIES DATA MANAGEMENT SERIES (DMS 1100 SYSTEM SUPPORT FUNCTIONS DATA ADMINISTRATOR REFERENCE MANUAL, UP-7909, Rev. 3, Sperry Univac, 1975.

[Ste76] Stemple, D. W. A database management facility for automatic generation of database managers. ACM TRANS. ON DATABASE SYSTEMS, Vol. 1, No. 1, March 1976, pp. 79-94.

[SWKH76] Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. ACM TRANS. ON DATABASE SYSTEMS, Vol. 1, No. 3, September 1976, pp. 189-222.

[THLZ77] Travis, L., Honda, M., LeBlanc, R., and Zeigler, S. Design rationale for TELOS, a PASCAL-based AI language. PROC. SYMPOSIUM ON ARTIFICIAL INTELLIGENCE AND PROGRAMMING LANGUAGES, August 15-17, 1977, pp. 67-76.

[Tsi75] Tsichritzis, D. A network framework for relational implementation., in DATA BASE DESCRIPTION, B. C. Douque and G. M. Nijssen (Eds.), North Holand, Amersterdam, The Netherlands, 1975, pp. 269-282.

[Tsi76] Tsichritzis, D. LSL; a link and selector language. ACM-SIGMOD INTERNAT. CONF. ON MANAGEMENT OF DATA. June 1976, Washington, D. C., pp 123-134.

[Weg74] Wegbreit, B. The treatment of data types in EL1. COMM. ACM, Vol. 17, No. 5, May 1974, pp. 251-265. [Wen75] Wensley, J. H. The impact of electronic disks on system architecture. COMPUTER, Vol. 8, No. 2, February 1975, pp. 44-48.

[Wor77] Wortman, D. B. (Ed.). PROC. OF THE ACM CONFERENCE ON LANGUAGE DESIGN FOR RELIABLE SOFTWARE. SIGPLAN NOTICES, Vol. 12, No. 3, March 1977.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 1970	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Technical
4. TITLE (and Subtitle): The Design and Implementation of a Database Management System Using Abstract Data Types	5. TYPE OF REPORT & PERIOD COVERED: Summary Report, no specific reporting period
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): A. James Baroody, Jr. and David J. DeWitt	8. CONTRACT OR GRANT NUMBER(s): DAAG29-75-C-0024 MCS78-01721
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin, 610 Walnut Street, Madison, Wisconsin 53706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 8 - Computer Science
11. CONTROLLING OFFICE NAME AND ADDRESS: See Item 18 below	12. REPORT DATE: June 1979
	13. NUMBER OF PAGES: 52
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D.C. 20550

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database management systems, abstract data types, generic procedures, CODASYL, query execution

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design, implementation, and performance analysis of a database management system implemented using abstract data types are presented. The use of abstract data types as an implementation tool is shown to have several significant advantages over current implementation techniques. First, by using a combination of abstract data types and generic procedures to structure the design of a database system, the resulting software will be more reliable. Also, by employing a programming language which supports specification and verification of abstract data types, we can guarantee data independence. Finally, we demonstrate that the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (continued)

application of abstract data types permits the elimination of run-time interpretation of the schema and subschema such as in IBM's IMS, Univac's DMS1100, and INGRES. Instead, the data manipulation routines, which are shown to be examples of generic procedures, are implemented as parameterized calls to the procedures bound to the instances of the three abstract data types used to represent the logical structure of the database.